Temporal Prolog

AUTHOR(S):

SAKURAGAWA, Takashi

CITATION:

SAKURAGAWA, Takashi. Temporal Prolog. 数理解析研究所講究録 1986, 586: 305-329

ISSUE DATE:

1986-03

URL:

http://hdl.handle.net/2433/99379

数理解析研究所講究録
第586巻 1986年 305-329

Temporal Prolog

Takashi SAKURAGAWA
Research Institute for Mathematical Sciences
Kyoto University
Kitashirakawa Sakyo-ku
Kyoto 606, Japan

Abstract

Temporal Prolog is a logic programming language based on temporal logic. This language allows clear and natural representation of state-transition, concurrency, mutual exclusion and nondeterminism. A formal semantics of the language is given. As an application, its use as an executable specification language for real time systems is presented.

1 Introduction

There are at least two purposes for which concurrency in programs is desired. One is to execute programs faster. One expects faster parallel computation when he writes a program in a concurrent language than in a sequential one because it is difficult to execute in parallel a sequentially written program in an effective manner. The other is to provide programs with natural structure. If we write a simulation program in a sequential way, the program becomes clumsy when the phenomenon simulated occurs in parallel in the real world.

It depends on application and requirement which of the purposes is more important. The first category consists of applications like data processing in which high speed computation is desired. Requirement of faster computation is often the motive of describing a data processing program in a concurrent manner. A characteristic of the first category is that we seldom take care of the order of computation and transition of internal states of processes. For example, to sort a huge amount of data, we want to execute the program in parallel and get the result faster. We do not want to know internal states of processes except for debugging purposes. This becomes clearer supposing that we write such a program like sort in a stream programming language. Generally, state transition is sometimes not essential for the first category.

On the other hand, explicit inclusion of the notion of state transition is essential for such systems as simulation, man-machine interaction, data bases and real-time systems. The second category consists of these applications. It is not possible to write programs which simulates the phenomena occurring in parallel in the real world without taking account of the state transition of each process. Similarly, real time program is developed considering the internal states of each component of the system. For the second category, a language in which concurrency and state transition can be naturally expressed is suitable.

Object oriented programming languages like SIMULA or smalltalk are suited for simulating phenomena in the real world because those notions like "internal state" and its transition along the "time" axis are included in those languages.

At present we have several concurrent logic programming languages which were designed in order to describe concurrent computation: Concurrent Prolog (C.P.) [Shapiro1983] and PARLOG. In those languages, concurrent computation stands for concurrent resolution. Internal states and their transitions are regarded as states of resolution and their transitions.

Because the original logical system (i.e. first order logic) does not include the notion of time, the state transitions are not expressed explicitly in programs. Therefore, C.P. is rather suited for the first category applications. For the second category, however, it sometimes becomes a disadvantage. In C.P., in order to specify the way of state transitions of processes read only annotation and commit operator must be introduced, which do not exist in the original logical system.

If we design a logic programming language based on a logical system which includes the notion of time, we can describe the state transition in the original system itself. From this point of view, we choose temporal logic as a basis.

In this paper, a concurrent programming language "Temporal Prolog" is presented. In *section 3*, we present the informal syntax of Temporal Prolog. In *section 4*, we describe several simple examples in order to explain what expression is possible in Temporal Prolog. After that, we give a formal semantics of Temporal Prolog in *section 5*.

2 Temporal Logic

Temporal logic is an extension of classical logic. In order to deal with the notion of time, some modal operators are added.

For example,

$a \Rightarrow \bigcirc b$
$a \Rightarrow \Box b$
$a \Rightarrow \Diamond b$
$a \Rightarrow b$ **until** c

Each formula has the following informal meaning respectively.

If a is true at some point in time
then b is true at the next point in time.
If a is true at some point in time
then b will be true forever from that point in time.
If a is true at some point in time
then b will become true at some time in the future including that point in time.
If a is true at some point in time
then b will be true until c becomes true.

Usually, we use these four symbols:○□◇**until**, in addition to the symbols used in first order logic (we introduce other modal operators in this paper).

3 Temporal Prolog

We do not present the formal syntax of Temporal Prolog completely to parse a program by computers, but present only an adequate informal syntax for human beings.

First of all, we define the sets of symbols.

V: the set of variables
SF: the set of skolem functions
PFi: the set of internal pattern functions
PFe: the set of external pattern functions
PF = PFi ∪ PFe
Pi: the set of internal predicates
Pe: the set of external predicates $=, true, false, at \in$ Pe
P = Pi ∪ Pe

$at(n)$ is true iff the program was executed just n steps.

V,SF,PFi,PFe,Pi and Pe are disjoint. Each element of SF,PFi and PFe has a non negative integer (arity) which represents the number of arguments.

PFi and Pi are pattern functions and predicates which are defined in the program, PFe and Pe are pattern functions and predicates which are defined outside of the program (i.e. given functions and predicates).

Constants are skolem functions whose arities are naught.

We define term and atomic formula.

T(term)

$V \subseteq T$
$f \in SF \cup PF$, arity of f is k, $t_1 \ldots t_k \in T$ implies $f(t_1, \ldots t_k) \in T$
$t \in T$, an element of PF is found in t implies $\bullet t \in T$

The value of $\bullet t$ is the value of t at the previous point in time.

AF(atomic formula)

$p \in P$, arity of p is k, $t_1 \ldots t_k \in T$ implies $p(t_1, \ldots t_k) \in AF$

AFi is the set of atomic formulas whose predicates are elements of Pi.

Next, we define condition formula.

CF(condition formula)

AF ⊆ CF

Let c,d ∈ CF, n be a non-negative integer.

~c ∈ CF	;not c
●c ∈ CF	;c was true at the previous point in time. (if there is no previous point in time, false.)
■c ∈ CF	;c has been true until now (including now).
◆c ∈ FS	;c was true at some point in the past including now.
c **since** d ∈ CF	;c has been true since d was true last.
c **after** d ∈ CF	;c became true at least once after d had become true.
c **for** n ∈ CF	;c was true for n times continuously. ;where n is a positive integer.
c ∧ d ∈ CF	;c and d

A legal program of Temporal Prolog is a subset of R, which satisfies the condition in *Section 5*.

R(result)

AFi ⊆ R
f ∈ PFi, arity of f is k, t_0 ... t_k ∈ T implies $f(t_1, \ldots t_k)$->t_0 ∈ R
;$f(t_1, \ldots t_k)$ is reducible to t_0.

Let q,r ∈ R, c ∈ CF .

c => r ∈ R	;c implies r.
q ∧ r ∈ R	;q and r.
□r ∈ R	;r is true forever.
r **until** c ∈ R	;r is true until c becomes true.
r **atnext** c ∈ R	;r becomes true when c first becomes true.

atnext was introduced in [Kröger1984].

Temporal Prolog includes pure Prolog.

Example

Let a,b,c,d ∈ Pi; f ∈ PFi; 0,1 ∈ SF;X ∈ V, then

~$a(f(X))$ ∧ (●$a(1)$ **since** ■$c(X)$) => ($f(X)$ -> 0) **until** d

is an element of R.

4 Simple Examples

In this section, we present simple programming examples and explain what descriptions are possible in Temporal Prolog.

4.1 Concurrency

To describe concurrent processes in Temporal Prolog, The only thing we must do is to write the processes in parallel. For instance, the following program controls two foot warmers concurrently.

$$\begin{aligned} &temperature\_of\_foot\_warmer1 > comfortable\_temperature \\ &\qquad \Rightarrow off\_foot\_warmer1 \\ &temperature\_of\_foot\_warmer1 < comfortable\_temperature \\ &\qquad \Rightarrow on\_foot\_warmer1 \end{aligned}$$

$$\begin{aligned} &temperature\_of\_foot\_warmer2 > comfortable\_temperature \\ &\qquad \Rightarrow off\_foot\_warmer2 \\ &temperature\_of\_foot\_warmer2 < comfortable\_temperature \\ &\qquad \Rightarrow on\_foot\_warmer2 \end{aligned}$$

4.2 Class and instance

Above two processes do the just same thing. We apply the notion of object oriented programming to it.

$$\begin{aligned} &temperature\_of\_foot\_warmer(X) > comfortable\_temperature \\ &\qquad \Rightarrow off\_foot\_warmer(X) \\ &temperature\_of\_foot\_warmer(X) < comfortable\_temperature \\ &\qquad \Rightarrow on\_foot\_warmer(X) \end{aligned}$$

This program defines the class "foot warmer controller". We call X instance variable because the value of X discriminates an instance from the others. If the domain of X is $\{1,2\}$ then this program is equivalent with the program in *4.1*.

4.3 Starting and Terminating processes

In the above program, processes control foot warmers from the first point in time and will control them forever. If we want to control a foot warmer from some point in time, the following program is adequate.

$$\begin{aligned} start(X) \Rightarrow \Box(&(temperature\_of\_foot\_warmer(X) > comfortable\_temperature \\ &\qquad \Rightarrow off\_foot\_warmer(X)) \wedge \\ &(temperature\_of\_foot\_warmer(X) < comfortable\_temperature \\ &\qquad \Rightarrow on\_foot\_warmer(X))) \end{aligned}$$

If *start*(*n*) becomes true at some point in time, a process whose instance variable's value is *n* is invoked. This process controls foot warmer*n* forever.

Furthermore, to terminate a process at some point in time,

start(*X*) => ((*temperature_of_foot_warmer*(*X*) > *comfortable_temperature*
=> *off_foot_warmer*(*X*)) ∧
(*temperature_of_foot_warmer*(*X*) < *comfortable_temperature*
=> *on_foot_warmer*(*X*))) **until** *end*(*X*)

start(*n*) invokes an instance and *end*(*n*) terminates an instance whose instance variable's value is *n*.

until can play the roles of *assert*,*retract*.

assert => (*A* => *B*) **until** *retract*

If *assert* becomes true, *A* => *B* is asserted. After that, if *retract* becomes true, *A* => *B* is deleted.

4.4 Inter process communication

There are two types of communication in Temporal Prolog. One is to refer the internal states of other processes without disturbing them. Another is to send a message to other processes and change their internal states. (We did not define the "process" nor its internal states until now. A Process is a set of instances of atomic formulas. Its state is their truth values. For instance, {*off_foot_warmer*(*1*),*on_foot_warmer*(*1*)} is a process and {*off_foot_warmer*(*2*),*on_foot_warmer*(*2*)} is another process in *4.2*. Of course we can define above two processes as one process. There is a lot of freedom to define a process. Although this notion is different from ordinary one, it plays the role of process in Temporal Prolog.)

The following program is a temperature monitor, which gives an alarm if one of the one hundred thermometers indicates more than one hundred degree.

temp_is(*X*,*C*) ∧ *C*>*100* => *dangerous*(*X*)
Process1 (in fact, one hundred processes)

dangerous(*X*) => □*alarm*
Process2

The domain of *X* is {1, ... 100}.

Process1 watches the temperature and process2 gives an alarm.

Let process1 {*temp_is*(*n*,*C*),*dangerous*(*n*)} and process2 be {*alarm*}. Process1 does not change the internal state of process2. Process2 watches the internal state of process1

(of course, without changing the state of process1).

On the other hand, if we define process1 as {*temp_is*(*n*,*C*)} and process2 as {*dangerous*(*n*),*alarm*}. Then process1 sends a message to process2 and changes its state.

Due to the definition of processes, We get different interpretations of process communication.

Inter process communication in Temporal Prolog is quite different from stream programming languages.

4.5 Wait

atnext enables a process to wait a signal.

```
    .
    .
... => wait
wait => restart atnext signal
restart => continuation
    .
    .
```

When it becomes necessary to synchronize with another process, *wait* becomes true. After that, at the first time when another process makes *signal* true, *restart* becomes true and process1 continues its execution.

4.6 Mutual exclusion

Next example is mutual exclusion program of one resource.

```
assign(1) ∧ ●assigned_to(1) => assigned_to(1)
assign(2) ∧ ●assigned_to(2) => assigned_to(2)
assign(1) ∧ ●~assigned_to_something => assigned_to(1)
assign(2) ∧ ~assign(1) => assigned_to(2)
assigned_to(X) => assigned_to_something
```

The domain of X is {1,2}

(first and second lines can be replaced by "*assign*(*X*) ∧ ●*assigned_to*(*X*) => *assigned_to*(*X*)".)

Process *n* makes *assign*(*n*) true when it wants to use the resource. That process waits until *assigned_to*(*n*) becomes true by the program in section *4.5*. Of course, the process *n* must keep *assign*(*n*) true until this resource becomes not necessary for process *n*. For this action, the use of **until** in section *4.3* is available.

In the above example, process 1 has a higher precedence than process 2. (i.e. if process 1 and process 2 want to use this resource simultaneously when it is unused, process 1 gets it.)

4.7 Nondeterminism

The example in section *4.6* is deterministic. That program is suitable if actually we would like to give precedences to processes. On the other hand, sometimes we do not care which process gets the resource when plural processes want to get a resource simultaneously.

$assign(X) \wedge \bullet assigned\_to(X) \Rightarrow assigned\_to(X)$
$assign(X) \wedge \sim assigned\_to\_another(X) \Rightarrow assigned\_to(X)$
$assigned\_to(X) \wedge \sim X{=}Y \Rightarrow assigned\_to\_another(Y)$

The domain of X is {1,2}.

$assigned\_to\_another(n)$ is true when the resource is assigned to some process other than process n.

When process 1 and process 2 want to be assigned the resource simultaneously, it is assigned nondeterministically. (i.e. the resource will be assigned to process 1 or it will be assigned to process 2.)

note: Above program can be applied even if the domain of X is changed.

5 Semantics

We define the semantics of Temporal Prolog by a transformation. We transform a program in Temporal Prolog to special formulas (we call them *normal* formula) and define the semantics of normal formulas as a program.

The advantage of this approach is that we can easily and concisely define the formal semantics. Furthermore, we can convert the normal formulas into Prolog after the transformation and automatically get an implementation of Temporal Prolog although the aim of the transformation is just to present a formal semantics.

We define the formal semantics of normal formulas in the same way as pure Prolog [Apt,Emden1982]: we give a sort of minimal model as a meaning of normal formulas. The execution of normal formulas is to construct the model, i.e. to decide an instance of atomic formula is true or false in the model.

From now on we treat $\wedge$ as n-ary operator. In other words, $a \wedge (b \wedge c)$ and $(a \wedge b) \wedge c$ are the same formula as $a \wedge b \wedge c$. Similarly, we do not distinguish $A \Rightarrow (B \Rightarrow C)$ and $A \wedge B \Rightarrow C$. Therefore, $a \wedge ((b \wedge c) \wedge d) \Rightarrow ((e \wedge f) \Rightarrow g)$ is treated as if that is $a \wedge b \wedge c \wedge d \wedge e \wedge f \Rightarrow g$.

As described Above, a program of Temporal Prolog is transformed to normal formulas. Normal formula is like the following:

$c_1 \wedge c_2 \ldots \wedge c_k \Rightarrow d$

where $d \in$ AFi, c_j is ● ... ●a or ● ... ●~a, $a \in$ AF (of course, an atomic formula and an atomic formula with a negation are legal). There is no pattern function and no modal operator in c_j.

Examples

Let $a,b,c,d \in$ P, $f \in$ SF, $X \in$ V.

●$b(X) \wedge$ ~$c(X)$ => $d(f(X),Y)$

is a normal formula. While the next is not.

~●$c(X)$ => a

5.1 Algorithm of transformation

In this section, we denote elements of AF by a and b, elements of R by q,r and s, an *new* element of Pi by p respectively. $fv(a)$,$fv(r)$ and $fv(a,r)$ means the free variables in a, r and the union of $fv(a)$ and $fv(r)$ respectively.

Step 1

In this step we eliminate □,**until** and **atnext**.

(Termination condition) Each formula in Pr takes the form r or a => r and r is an atomic formula or $f(t_1, \ldots, t_k)$ -> t_0. Where Pr is the program which is transformed.

If the condition is not satisfied, we take a formula, an element of Pr, s.t. because of its existence, the termination condition is not satisfied, and transform it in the following way.

(1) $q \wedge s$ --> q
s

a => $q \wedge s$ --> a => q
a => s

(2) □q --> q

a => □q --> a => $p(X_1, \ldots, X_k)$
●$p(X_1, \ldots, X_k)$ => $p(X_1, \ldots, X_K)$
$p(X_1, \ldots, X_k)$ => q
where $X_1, \ldots, X_k$ are $fv(q)$

(3) q **until** a --> ~a => q

a => q **until** b --> a => $p(X_1, \ldots, X_k)$
$\bullet p(X_1, \ldots, X_k) \wedge \bullet \sim b$ => $p(X_1, \ldots, X_k)$
$p(X_1, \ldots, X_k) \wedge \sim b$ => q
where $X_1, \ldots, X_k$ are $fv(q,b)$

(4) q **atnext** a --> a => q

a => q **atnext** b --> a => $p(X_1, \ldots, X_k)$
$\bullet p(X_1, \ldots, X_k) \wedge \bullet \sim b$ => $p(X_1, \ldots, X_k)$
$p(X_1, \ldots, X_k) \wedge b$ => q
where $X_1, \ldots, X_k$ are $fv(q,b)$

Repeat the above procedure until (termination condition) is satisfied. This repetition will stop eventually because the number of □, **until**, **atnext** and ∧(in the consequences) decrease one by one. We get the unique result except the difference of new predicates names even if we change the order of the transformation.

Example

$(temp1+temp2)/2>120$ **for** 3 => $switch\_off \wedge \Box alarm$

is transformed to normal formulas in the following way.

$(temp1+temp2)/2>120$ **for** 3 => $switch\_off$
$(temp1+temp2)/2>120$ **for** 3 => $\Box alarm$

$(temp1+temp2)/2>120$ **for** 3 => $switch\_off$
$(temp1+temp2)/2>120$ **for** 3 => p
$\bullet p$ => p
p => $alarm$

Step 2

We eliminate **since**, **after**, **for**, ■ and ◆ in Pr.

(Termination condition) There is no **since**, **after**, **for**, ■ nor ◆ in Pr.

If the condition is not satisfied, we take a formula from Pr and in which there is **since**, **after**, **for**, ■ or ◆, and transform it in the following way.

In the next table, ... **X** ... => r means an element of Pr and **X** is a condition formula which has just one modal operator except ●.

(1) ... ■a ... => r --> ... $p(X_1, \ldots, X_k)$ ... => r
$a \wedge at(0)$ => $p(X_1, \ldots, X_k)$

$a \wedge \bullet p(X_1, \ldots, X_k) \Rightarrow p(X_1, \ldots, X_k)$
where $X_1, \ldots, X_k$ are $fv(a)$

(2) $\ldots \blacklozenge a \ldots \Rightarrow r$ --> $\ldots p(X_1, \ldots, X_k) \ldots \Rightarrow r$
$a \Rightarrow p(X_1, \ldots, X_k)$
$\bullet p(X_1, \ldots, X_k) \Rightarrow p(X_1, \ldots, X_k)$
where $X_1, \ldots, X_k$ are $fv(a)$

(3) $\ldots a$ **since** $b \ldots \Rightarrow r$ --> $\ldots p(X_1, \ldots, X_k) \ldots \Rightarrow r$
$b \wedge a \Rightarrow p(X_1, \ldots, X_k)$
$\bullet p(X_1, \ldots, X_k) \wedge a \Rightarrow p(X_1, \ldots, X_k)$
where $X_1, \ldots, X_k$ are $fv(a,b)$

(4) $\ldots a$ **after** $b \ldots \Rightarrow r$ --> $\ldots p(X_1, \ldots, X_k) \ldots \Rightarrow r$
$a \Rightarrow p(X_1, \ldots, X_k)$
$\bullet p(X_1, \ldots, X_k) \wedge \sim b \Rightarrow p(X_1, \ldots, X_k)$
where $X_1, \ldots, X_k$ are $fv(a,b)$

(5) $\ldots a$ **for** $n \ldots \Rightarrow r$ --> $\ldots (a \wedge \bullet a \wedge \ldots \wedge \bullet^{n-1} a) \ldots \Rightarrow r$
;An alternative is to generate a predicate defined recursively.

Repeat the above until the (termination condition) is satisfied. This transformation will stop eventually because the number of **since**, **after**, **for**, ■ and ◆ in Pr decrease one by one. We get the unique result except the difference of new predicates names even if we change the order of transformations of formulas. After this step, we have no modal operators in Pr except ●.

Example

The program:

(temp1-temp2)/2>120 **for** *3* => *switch_off*
(temp1-temp2)/2>120 **for** *3* => *p*
●*p* => *p*
p => *alarm*

is transformed to:

(temp1-temp2)/2>120 ∧ ●(*(temp1-temp2)/2>120*) ∧ ●●(*(temp1-temp2)/2>120*)
=> *switch_off*
(temp1-temp2)/2>120 ∧ ●(*(temp1-temp2)/2>120*) ∧ ●●(*(temp1-temp2)/2>120*)
=> *p*
●*p* => *p*
p => *alarm*

Step 3

This step is consist of two substeps. In the first substep, we introduce new internal predicates which are correspond to internal pattern functions. Second substep is expansion of pattern functions.

First substep
(Termination condition) There is no -> in Pr.

We chose a formula whose form is $f(t_1, \ldots, t_k) \rightarrow t_0$ or $a \Rightarrow f(t_1, \ldots, t_k) \rightarrow t_0$ and convert it in the following way.

$$f(t_1, \ldots, t_k) \rightarrow t_0 \quad \longrightarrow \quad f(t_1, \ldots, t_k, t_0)$$

$$a \Rightarrow f(t_1, \ldots, t_k) \rightarrow t_0 \quad \longrightarrow \quad a \Rightarrow f(t_1, \ldots, t_k, t_0)$$

We add f to Pi as a new internal predicate whose arity is the arity of f plus 1.

Second substep
(Termination condition) There is no pattern function in Pr.

Let f is an occurrence of pattern function which is most outside (i.e. does not occur in an argument of another occurrence of a pattern function) and occurs right most. Its expansion is done in the following way.

$$.. \; p(\ldots f(t_1, \ldots, t_k) \ldots) \ldots\ldots \Rightarrow r$$
$$\longrightarrow \quad .. \; (p(\ldots X \ldots) \wedge \bullet^n f(t_1, \ldots, t_k, X)) \ldots\ldots \Rightarrow r$$

$$.. \Rightarrow p(\ldots f(t_1, \ldots, t_k) \ldots)$$
$$\longrightarrow \quad .. \wedge \bullet^n f(t_1, \ldots, t_k, X) \Rightarrow p(\ldots X \ldots)$$

Where p is a predicate, X is a new variable. n is the number of ●s which occur between levels of p and f. For example, n is 2 for f and 3 for u in $p(\bullet b(X, \bullet\bullet u, \bullet f(2)), Y)$.

According to f is an external or internal pattern function, we add f to Pe or Pi as an external or internal predicate whose arity is incremented by one.

After the second substep, we eliminate the all ●s which are attached to terms.

Examples

$append([], X) \rightarrow X$
$append([A|X], Y) \rightarrow [A|append(X, Y)]$

is converted to:

$append([],X,X)$
$append([A|X],Y,[A|append(X,Y)])$

$append([],X,X)$
$append(X,Y,Z) \Rightarrow append([A|X],Y,[A|Z])$

$0+X \rightarrow X$
$s(X)+Y \rightarrow s(X+Y)$

is converted to:

$+(0,X,X)$
$+(s(X),Y,s(X+Y))$

$+(0,X,X)$
$+(X,Y,Z) \Rightarrow +(s(X),Y,Z)$

$sum \rightarrow temp+\bullet temp$

is converted to:

$sum(temp+\bullet temp)$

$+(temp,\bullet temp,X) \Rightarrow sum(X)$

$\bullet temp(Y) \wedge +(temp,Y,X) \Rightarrow sum(X)$

$\bullet temp(Y) \wedge temp(Z) \wedge +(Z,Y,X) \Rightarrow sum(X)$

Step 3 is essentially the same as *expansion* in [Tamaki1984]. The difference is we deal with ●.

Step 4

After this step, each negation is directly attached to an atomic formula in Pr.

(Termination condition) Each negation is directly attached to an atomic formula in Pr.

$\ldots \sim a \ldots \Rightarrow r \quad \rightarrow \quad \ldots \sim p(X_1, \ldots, X_k) \ldots \Rightarrow r$
$a \Rightarrow p(X_1, \ldots, X_k)$

where $X_1, \ldots, X_k$ are $fv(a)$
a is not an atomic formula.

Repeat the above procedure until the (termination condition) is satisfied.

This transformation will stop eventually too. We get the unique result except the difference of new predicates names even if we change the order of selection of formulas.

Example

$\sim\bullet c \Rightarrow d$ --> $\sim e \Rightarrow d$, $\bullet c \Rightarrow e$

Step 5

Distribution of $\bullet$.

Example

$\bullet(a \wedge \bullet b) \Rightarrow r$ --> $\bullet a \wedge \bullet\bullet b \Rightarrow r$

After these 5 steps, we get a transformed program Pr in which all the formulas are normal.

5.2 Model

While a model of a set of formulas in ordinary first order logic defines the meaning of functions and predicates just in one world. A model of a set of formulas in temporal logic is an infinite sequence of worlds (time) where the domain, the interpretations of functions and predicates, i.e. values of functions and truth values of predicates, can vary in time.

In this paper, we do not deal with such a general model. The domain is always the Herbrand Universe of A. The interpretations of functions are data constructors similarly to pure Prolog and do not vary. The only thing which can vary in time is the interpretation of each predicate.

The meaning of = is the equality in Herbrand Universe.

$\bigcirc \rightarrow \bigcirc \rightarrow \bigcirc \rightarrow \bigcirc \rightarrow$
$w(0)\ w(1)\ w(2)\ w(3)$

Model of temporal logic

In this paper, we denotes the n'th world by $w(n)$.

Like first order logic, the truth value of a formula is automatically defined if

the interpretations of domains, functions and atomic formulas are defined. The difference is the truth value of a formula is given in each world while only one truth value is given to a formula in first order logic.

The truth value of a formula in (this paper's version of) temporal logic is defined in the following way.

Let F be a formula of temporal logic.

$\bullet F$ at $w(n)$ iff $n>0$ and F at $w(n-1)$

~ ∧ ∨ => <=> ∀ ∃ are defined similarly to first order logic at each world. We do not define the meaning of ○,□,◇,■,◆,**atnext**,**until**,**since**,**after**,**for** because their meaning are not necessary to define the semantics of normal formulas.

Because We have already defined the domain and the interpretations of functions, we can define a model of A by giving interpretations of ground instances of atomic formulas in all $w(n)$. Therefore, we regard $w(n)$ as $\{p(d_1, \ldots, d_k);\ p(d_1, \ldots, d_k)$ at $w(n)\}$.

$W(A) = \{p(d_1, \ldots, d_k);\ p$ is a predicate in A, $d_1, \ldots, d_k \in$ domain$\}$

$w(n)$ is a subset of $W(A)$.

The exact definition of the model M of A is $\langle w(0), w(1), \ldots \rangle$.

A tuple $\langle W_0, \ldots, W_k \rangle$ is a division of $W(A)$ iff $W_0, \ldots, W_k$ are disjoint and $\bigcup W_i = W(A)$.

Let $\langle W_0, \ldots, W_k \rangle$ be a division of $W(A)$, L:$\langle v(0), v(1), \ldots \rangle$ and M:$\langle w(0), w(1), \ldots \rangle$ be models of A. L>M iff there are non-negative integers m,n s.t.

for all non-negative integer l, $l<m$ implies $v(l)=w(l)$ and
for all non-negative integer i, $i<n$ implies $v(m)\cap W_i = w(m)\cap W_i$ and
$w(m)\cap W_n \subsetneq v(m)\cap W_n$.

$>$ is a partial order relation among the models of A.

Let M be a model of A.
M satisfies A iff any formula in A is true at any point in time.
M is minimal iff M is minimal by $>$ among the all models in which all formulas in A is always true.
M is least iff M is the unique minimal model.

5.3 Model construction

In this section, we give a formal semantics of a program Pr in which all formulas are normal. We quantify all free variables which occur in a formula at outside

of it. (We abbreviate these quantifiers.) The interpretation of external predicates are already defined (i.e. their meaning are given externally).

We give several definitions.

Dependency relation

First, we eliminate all atomic formulas to which a ● is attached.

Example

$\bullet a \wedge \bullet\bullet \sim b \wedge c \wedge \sim d \Rightarrow e$

becomes

$c \wedge \sim d \Rightarrow e$

We apply this elimination to all formulas in Pr.

Dependency relation is defined in the following way.

For all $p,q \in$ Pr,

$d^{+}(p,q)$	iff	A formula whose form is ... $\wedge$ q(...) $\wedge$ ... => p(...) occurs in Pr.
$d^{-}(p,q)$	iff	A formula whose form is ... $\wedge$ ~q(...) $\wedge$ ... => p(...) occurs in Pr.
$d(p,q)$	iff	$d^{+}(p,q)$ or $d^{-}(p,q)$

We get a partial preorder d^{*} by extending the relation d to a transitive relation and define an equivalence relation $d^{=}$ which is obtained by d^{*}. We denote an equivalence class of p by $[p]$. d^{*} is applicable to $[p]$ naturally.

We divide the internal predicates in Pr to equivalent classes $[p_1]$, ... ,$[p_n]$, provided that $i<j$ implies not $d^{*}([p_i], [p_j])$ (condition 0). We define $[p_0]$ as {all external predicates} and W_i as $\{p(d_1, \ldots, d_k); p \in [p_i], d_j \in$ domain, k=arity of $p\}$. $\langle W_0, \ldots, W_n \rangle$ is a division of W(Pr).

If there is no pair of $p,q \in [p_i]$ s.t. $d^{-}(p,q)$ for all i, $0 \leq i \leq k$(condition 1), then we get the following theorem. A program in Temporal Prolog is legal if the condition 1 holds after the transformation described in *5.1.*

Theorem 1
If the condition 1 is satisfied then there is the least model of Pr.

(proof)

We define $f_{pq}:P(W(Pr))^p \times P(W_0) \times \ldots \times P(W_q) \to P(W_q)$ for $0 \leq p$, $1 \leq q \leq n$ in the following way. (where $P(A)$ denotes the power set of A.)

$c_0 \in f_{pq}(w_0, \ldots, w_{p-1}, x_0, \ldots, x_q)$ iff

there is an instance of a formula in Pr s.t.

$\bullet^{m_1}(\sim)c_1 \wedge \ldots \wedge \bullet^{m_k}(\sim)c_k \Rightarrow c_0$ where

$0 \leq m_i \leq p$

$c_i \in w_{p-m_i}$ if $m_i \neq 0$ and $\sim$ is not attached to c_i.

$c_i \notin w_{p-m_i}$ if $m_i \neq 0$ and $\sim$ is attached to c_i.

$c_i \in \bigcup_{i=0,q} x_i$ if $m_i = 0$ and $\sim$ is not attached to c_i.

$c_i \notin \bigcup_{i=0,q} x_i$ if $m_i = 0$ and $\sim$ is attached to c_i.

for all i $1 \leq i \leq k$

$\lambda x \,.\, f_{pq}(w_0, \ldots, w_{p-1}, x_0, \ldots x_{q-1}, x)$ is continuous when we consider $P(W_q)$ lattice ordered by $\subseteq$ because the condition 1 is satisfied.

Now, we construct the least model of Pr.

We denote $w_p \cap W_q$ by w_{pq}. If w_{pq} is defined for all q $0 \leq q \leq n$, w_p is defined. We construct w_{pq} inductively.

Suppose $w_0 \ldots w_{p-1}$ are already defined. w_{q0} is also already defined because it consists of external predicates. w_{pq} is defined as the least fixed point of

$\lambda x \,.\, f_{pq}(w_0, \ldots, w_{p-1}, w_{p0}, \ldots w_{pq-1}, x)$

inductively.

$\langle w_0, w_1, \ldots \rangle$ is a model of Pr.

If not, there must be a counter example, i.e. there is an instance of a formula in Pr, non-negative integers p and q s.t.

$\bullet^{m_1}(\sim)c_1 \wedge \ldots \wedge \bullet^{m_k}(\sim)c_k \Rightarrow c_0$ where

$0 \leq m_i \leq p$

$c_i \in w_{p-m_i}$ if $m_i \neq 0$ and $\sim$ is not attached to c_i.

$c_i \notin w_{p-m_i}$ if $m_i \neq 0$ and $\sim$ is attached to c_i.

$c_i \in \bigcup_{i=0,q} x_i$ if $m_i = 0$ and $\sim$ is not attached to c_i.

$c_i \notin \bigcup_{i=0,q} x_i$ if $m_i = 0$ and $\sim$ is attached to c_i.

for all i $1 \leq i \leq k$ and

$c_0 \notin w_{pq}$

the predicate of $c_0 \in [p_q]$

This contradicts with the way of construction of w_{pq}. (The least fixed point w_{pq} must contains such an instance.)

$\langle w_0, w_1, \ldots \rangle$ is the least model of Pr. because we make w_{pq} least to satisfy Pr in each inductive step.

□

If the condition 1 is satisfied then the least model is the semantics of Pr.

Even if there are several divisions which satisfy condition 0, the least model is defined uniquely.

Let Pr' be the formulas which are obtained by rewriting the formulas in Pr in the following way. First, we change the variable names, so that there is no variable which occurs in more than one formula. Second, we select a internal predicate, say p, and gather all the formulas whose forms are A => $p(...)$ or $p(...)$.

$$A_1 \Rightarrow p(t_{11}, \ldots, t_{1k})$$
$$\vdots$$
$$A_i \Rightarrow p(t_{i1}, \ldots, t_{ik})$$

(A_j can be empty.)

We get the following formula from the above formulas.

$$\begin{aligned}
&((\exists(Y_{11}, \ldots, Y_{1m_1}) \\
&\qquad (A_1 \wedge X_1{=}t_{11} \wedge \ldots \wedge X_k{=}t_{1k})) \vee \\
&\qquad \vdots \\
&(\exists(Y_{i1}, \ldots, Y_{im_i}) \\
&\qquad (A_i \wedge X_1{=}t_{i1} \wedge \ldots \wedge X_k{=}t_{ik}))) \\
&\Leftrightarrow \quad p(X_1, \ldots, X_k)
\end{aligned}$$

where $Y_{j1}, \ldots, Y_{jm_j}$ are the free variables which occurs in A_j and $p(t_{j1}, \ldots, t_{jk})$. $X_1, \ldots, X_k$ are new variables.

We apply this procedure to all internal predicates and get Pr'.

Theorem 2
If Pr satisfies condition 1 then the least model M satisfies Pr'.

(proof)

=> Because M is the model of Pr.

<= If not, there are a tuple $\langle d_1, \ldots, d_k \rangle$ and a non-negative integer n s.t.
$p(d_1, \ldots, d_k) \in w_n$ and
for all j $1 \leq j \leq i$
$\sim(A_j \wedge d_1{=}t_{j1}, \ldots, d_k{=}t_{jk})$
for all instantiations of $fv(A_j, p(t_{j1}, \ldots, t_{jk}))$

This contradicts with the way of construction of the least model.

□

We can verify a program by this theorem if we prepare an appropriate formal axiom system.

The condition 1 is not satisfied in the following example.

$$\sim a \Rightarrow a$$

Even in this case, there is at least one minimal model. A minimal model always exists because the model in which all instance of atomic formulas are true satisfies Pr. However,

$$\sim a \Leftrightarrow a$$

has no model. On the other hand,

$$\sim b \Rightarrow a$$
$$\sim a \Rightarrow b$$

has a model which satisfies the following formulas.

$$\sim b \Leftrightarrow a$$
$$\sim a \Leftrightarrow b$$

Therefore we have two ways of giving the semantics to a program which does not satisfies the condition 1.

(1) We give the semantics as all minimal models. (If there are plural minimal models, we can execute it according to any minimal model, this means the program is nondeterministic.) Any program which satisfies conditions in *section 4* is legal.

(2) If there is a model which satisfies Pr', Pr is a legal program. The semantics of Pr is defined as all minimal models of Pr'.

In this paper, we select (2). Therefore, we must decide whether Pr' can be satisfied or not, even if whatever interpretations of external predicates are given.

If the Herbrand Universe is finite, there is a sufficient condition which is decidable.

Each formula in Pr' can be converted to an equivalent propositional temporal logic formula if the Herbrand Universe is finite. Therefore, we consider the case there is no variable in A, a set of formulas of temporal logic, and suppose that there is no other temporal operator than ● in A.

We say a finite sequence of worlds $\langle w_0, \ldots, w_k \rangle$ $(0 \leq k)$ satisfies A iff each formula in A is true at w_k in the model $\langle w_0, \ldots, w_k, \phi, \phi, \ldots \rangle$.

We construct a nondeterministic finite automaton. The input alphabets are the states of external propositions (i.e. instances of external predicates) and consists of 2^m symbols, where m is the number of external propositions. The states of the automaton are elements of $\{0, \ldots n\} \times W(\mathrm{Pr})^{n+1}$, where n is the number of $\bullet$ which occur in A. $W(\mathrm{Pr})$ is regarded as {all the states of truth value of propositions} and finite. The initial state is $\langle 0, \phi, \ldots, \phi \rangle$. The state transition function δ is defined in the following way.

$\langle i, x_0, \ldots x_n \rangle \in \delta(\langle j, w_0, \ldots w_n \rangle, a)$ iff
$i = min(j+1, n)$ and
$x_k = w_{k+1}$ for all k $0 \leq k < n$ and
$a = x_n \cap W_0$ and
$\langle x_{n-j}, \ldots, x_n \rangle$ satisfies A.

where W_0 is the set of external propositions.

Now, we get a nondeterministic automaton. If the empty set does not appear as the value of δ for any input alphabet sequence, i.e. we can always construct the next world, A has a model. This condition is decidable.

The restriction, the Herbrand Universe is finite, seems too strong. In fact, it is possible to relax the restriction. We have not given a type to variables, functions nor predicates. i.e. there is only one type. When we type variables, functions and predicates, if "$t_1, \ldots t_k$ are types which occurs in $[p_i]$ in which there are p,q s.t. d-(p,q)" implies "the Herbrand Universe of types $t_1, \ldots t_k$ are finite", the above decision procedure is available. In real applications, the number of processes and resources are usually finite. Therefore, for example, mutual exclusion like in section *4.6* can be justified.

6 Implementation

We describe two ways of implementation in this section.

6.1 Transformation into Prolog

Supposing that we have closed formulas of temporal logic and its model, we can convert them to a closed formula of ordinary first order logic and its model so that the values of functions and truth values of predicates are naturally reserved, even if the domain and the interpretations of functions vary according to time. In this paper, We suppose that the domain and the interpretations of functions are fixed.

The following is the transformation.

By increasing the arities of predicates by one and we represent their truth values which vary according to time by their new arguments. For instance,

$a(X) \Rightarrow \Box p(b(X))$

is transformed to:

$a(W,X) \Rightarrow \forall W1(R(W,W1) \Rightarrow p(W1,b(X)))$

We call the variables W and $W1$ world variables. $R(W,W1)$ is a predicate which represents the accessibility. In temporal logic, the domain of world variables is non-negative integer and $R(W,W1)$ iff $W \leq W1$.

By transforming a program in which all formulas are normal, we get a (not pure) Prolog program. If there is no negation in the original program, we get a pure Prolog program.

Example

$dangerous(X) \Rightarrow p$
$\bullet p \Rightarrow p$
$p \Rightarrow alarm$

$dangerous(W,X) \Rightarrow p(W)$
$p(W) \Rightarrow p(s(W))$
$p(W) \Rightarrow alarm(W)$

where *dangerous* is an external predicate and s is the successor function.

The transformed program can be executed by ordinal Prolog interpreter or compiler. The condition 1 in *section 5.3*, however, must be satisfied and variables must be instantiated completely when a negation as failure is done [Clark1977].

6.2 Asserting the facts which are true at each point in time

The implementation described above has several disadvantages. In the above example, suppose that we would like to decide $alarm(1000)$ is true or not, then the interpreter calls p for 1000 times if *dangerous* has never become true. The speed of execution becomes slower and slower in time.

One solution of this problem is to assert the facts which is true at each point in time. Then we do not have to call p recursively 1000 times. On the other hand, exhaustive asserting is often redundant. For instance, we do not have to assert $alarm(n)$ in the above example. Because $alarm(n)$ does not occur in the condition part.

We divide the predicates into two category. Only the facts whose predicates are in the first category are asserted. In the above example, $\{p, dangerous\}$ is the first category and $\{p, alarm\}$ is the second category.

Only one $\bullet$ is attached to p and no $\bullet$ is attached to *dangerous*, we can retract the facts which describes about p at the two point before in time and the facts which describes about *dangerous* at the previous point in time.

If the condition 1 is not satisfied, we must extract a finite automaton and run

it.

7 *Application ... as an executable specification language*

One of the targets of Temporal Prolog is a use as an executable specification language.

The traditional way of building softwares for real time systems is the following: first, we write a (informal) specification, second, we design the program and implement according to it. Recently, an alternative is proposed [Zavel1984]: First, we define an abstract model of the system. Second, we write an executable specification on it. Finally, we transform the specification equivalently and get an efficient implementation.

This method seems it has several advantages: because the specification itself is executable, we can "debug" the specification by executing it. Prototyping is not necessary (because the specification itself plays the role of prototype). Furthermore, if the transformation for getting good efficiency is automated to some extent, we can easily cope with a change of specification even after we get an implementation.

The key of this method is the equivalent transformation. For the purpose of equivalent transformation, the semantics of the executable specification must be given strictly and simply. Otherwise, if not strict, we can not be sure the equivalence of two programs before and after the transformation. If not simple, probably the transformation becomes complex.

Another important factor is readability and writability of the executable specification language. For all practical purposes, it must be easily written and read by human beings. Otherwise, the specification in that language is just a program rather than specification.

Therefore, the design of the executable specification language is important. In [Jackson1983], a CSP like language is used. In [Zavel1982], a functional programming language is adopted.

We think Temporal Prolog is one of the candidates of executable specification language for real time systems because of its readability and its strict and simple semantics.

8 *Concluding remarks*

We proposed a concurrent logic programming language which includes the notion of time and state transition because the temporal logic, which is the basis of Temporal Prolog, includes such notions. Concurrency, mutual exclusion and nondeterminism can be easily expressed in this language.

By transforming a rather complex program into a simple one, we defined the formal semantics of Temporal Prolog.

We also described the implementation of Temporal Prolog. However, in order to enhance efficiency, further research is necessary. In order to apply this language to real

time controls, the research about the transformation which increase the efficiency of a program is also necessary.

Acknowledgement

The author would like to express his deep gratitude to Professor Reiji Nakajima for his appropriate advices. The author also thanks Mr. Masami Hagiya who let him know beneficial papers and Mr. Naruhiko Kawamura who read the earlier draft of this paper.

References

[Ajitomi1982]
N. Ajitomi, YAPS-Yet Another Programming System
Master thesis in computer science, Tokyo University (1982)

[Aoyagi,Fujita,Motooka1985]
T. Aoyagi, M. Fujita and T. Motooka
Tokio Kakeru Gengo (Time travelling language), in Japanese
Proceedings of the logic programming conference (1985) 8.1

[Apt,Emden1982]
K. R. Apt and M. H. van Emden, Contributions to Theory of Logic Programming
JACM Vol.29, No.3 (1982) pp.841-862

[Bellia,Degano,Levi1982]
M. Bellia, P. Degano and G. Levi,
The call by name Semantics of a Clause Language with Functions
Logic Programming, K. L. Clark and S. A. Tarnlund eds.,
pp.281-295, Academic Press (1982)

[Clark1977]
K. L. Clark, NEGATION AS FAILURE
LOGIC AND DATA BASES,
Gallaire and Minker eds., Plenum Press (1977) pp.293-322

[Emden,Kowalski1976]
M. H. van Emden and R. A. Kowalski,
The semantics of predicate logic as a programming language
JACM Vol.23, No.4 (1976)

[Fusaoka,Seki,Takahashi1984]
A. Fusaoka, H. Seki and K. Takahashi,
Description and Reasoning of VLSI Circuit in Temporal Logic

New Generation Computing, 2 (1984) pp.79-90

[Gabbay1976]
D. M. Gabbay, INVESTIGATIONS IN MODAL AND TENSE LOGICS
WITH APPLICATIONS TO PROBLEMS IN PHILOSOPHY AND LINGUISTICS
D. REIEL PUBLISHING COMPANY (1976)

[Goguen,Meseguer1984]
J. A. Goguen and J. Meseguer,
Equality, Types, Modules and Generics for Logic Programming

[Hagiya1984]
Theory of Modal Logic Programming
Software foundation 9-4 (1984), in Japanese

[Hagiya,Sakurai1984]
M. Hagiya and T. Sakurai,
Foundations of Logic Programming Based on Inductive Definition
New Generation Computing, 2 (1984) pp.59-77

[Hansson,Haridi,Tarnlund1982]
A. Hansson, S. Haridi and S.-A. Tarnlund,
Properties of a Logic Programming Language
Logic Programming, K. L. Clark and S. A. Tarnlund eds.,
pp.267-280, Academic Press (1982)

[Jackson1983]
M. A. Jackson, SYSTEM DEVELOPMENT
PRENTICE-HALL (1983)

[Kornfeld1983]
W. A. Kornfeld, Equality for Prolog
Proc. of IJCAI-VIII, pp.514-519

[Kröger1984]
F. Kröger, A Generalized Nexttime Operator in Temporal Logic
JOURNAL OF COMPUTER AND SYSTEM SCIENCES 29 (1984) pp.80-98

[Moszkowski1985]
B. Moszkowski, A Temporal Logic for Multilevel Reasoning about Hardware
COMPUTER February (1985) pp.10-19

[Mycroft,O'Keefe1984]
A. Mycroft and R. A. O'Keefe, A Polymorphic Type System for Prolog
Artificial Intelligence 23 (1984) pp.295-307

[Nakashima1984]
H. Nakashima, Term description
Proc. of the Logic Programming Conference '84, Tokyo (1984) 2-3, in Japanese

[Shapiro1983]
E. Y. Shapiro, A Subset of Concurrent Prolog and Its Interpreter
ICOT Tech. Report TR-003 (1983)

[Shibayama1984]
E. Shibayama, An extension of unification and its applications
in logic programming languages
Software foundation 10-4 (1984), in Japanese

[Tamaki1984]
H. Tamaki, SEMANTICS OF A LOGIC PROGRAMMING LANGUAGE
WITH A REDUCIBILITY PREDICATE
Proc. of International Symposium on LP, Atlantic City (1984)

[Tashiro,Senda,Miyakojima1983]
Tashiro,Senda,Miyakojima, A rule based system control method
Metrology and Control Vol.22 No.9 (1983) pp.42-46, in Japanese

[Wolper1981]
P. Wolper, TEMPORAL LOGIC CAN BE MORE EXPRESSIVE
IEEE 22nd Annual Symposium on Foundations of Computer Science

[Yonezaki,Nii,Hohrai1984]
Yonezaki, Nii, Hourai, Interval logic programming language: Templog
Proc. of first conference of Japan society for software science
and technology (1984)
1E-4 pp.77-80, in Japanese

[Zave1982]
P. Zave, An Operational Approach to Requirements Specification for Embedded
Systems
IEEE Trans. Software Engr. SE-8 (1982) pp.250-269

[Zave1984]
P. Zave, THE OPERATIONAL VERSUS, THE CONVENTIONAL APPROACH TO
SOFTWARE DEVELOPMENT
CACM Vol.27 No.2 (1984) pp.104-118